PigLatinJava
– troubleshooting examples

Marie Nordström

UMINF-07.26

ISSN-0348-0542

UMEÅ UNIVERSITY
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

PigLatinJava – troubleshooting examples

Marie Nordström
Dept. of Computing Science Umeå University S-901 87 Umeå, Sweden
marie@cs.umu.se

ABSTRACT

One of the problems teaching introductory object-oriented problem solving and programming is designing appropriate and enlightening examples and exercises. The type of examples we use in procedural programming is not necessarily adequate for illustrating similar syntactical concepts in an object-oriented paradigm. Moving to a higher level of abstraction, as with objects examples need to illustrate how solutions are to be designed and organised to uphold "good" object-oriented principles. Being aware of the factors influencing the success of an example illustrating a certain concept could aid in designing them. Critical aspects of examples are presented in this paper.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**] Object-oriented Programming
K.3.2 **[Computers and education]** Computer and Information Science Education - *Computer science education, Curriculum*

General Terms

Design

Keywords

Object orientation, objects-first, examples, CS1, UML, conceptual modelling, design, pedagogy, programming education

1 INTRODUCTION

When teaching object-oriented problem solving and programming the level of abstraction is much higher than in a procedural/imperative paradigm. Trying to introduce the paradigm objects-first adds extra constraints to the effort. Altogether this makes the design of examples/exercises (in the following called examples) much more demanding than in other paradigms.

One problem involved in this, is to show the difference between the model and the entity being modelled. The idea of decentralised responsibility makes objects have behaviours that the entity they model never would or could have. The use of "everyday things" in examples then becomes contradictory, leading to the wrong perception of responsibilities and behaviours of objects. Natural/physical objects seldom have abilities that we might consider "natural" from an object-oriented problem solving point of view.

Another problem is that objects tend to be "one-of-a-kind" in many examples. This is confusing, since we actually want to show the need for a class as a common descriptor of many objects. Furthermore, the difference between class and object becomes far from obvious. Making an example containing many objects requires an adequate context for the example to make sense. This easily makes the example too big in code and might even hide the features originally intended to show.

Being true to the objects-first approach adds to the difficulty, since the range of behaviour and modelling capabilities are limited, due to the lack of data structures and syntactical elements not yet introduced in the beginning of the course.

2 OBJECTS FIRST

Introducing problem solving and programming with an object-oriented paradigm/language has turned out to be more difficult than expected. Starting out teaching Java the "old" syntax-driven way, the results were alarming in terms of object orientation. Being painfully aware of this the CS-department initiated a research-project (FattaOO) in 2001 [6]. The objective of this project was to develop a design for the CS1 courses based on an object-oriented paradigm. Objects should be taught from the very beginning with an early focus on design and the teaching of general object-oriented concepts should be favoured over the syntactical details of Java. This project resulted in a set of guidelines for the development of an object-centric, holistic approach to the teaching of OO with Java [6].

This work was greatly influenced by the work done by Michael Kölling et.al. in the research group producing BlueJ [5,9].

2.1 Guidelines for objects-first

These are some of the guidelines developed:

2.1.1 No magic

Every topic seeming to involve superior forces (from the student point of view) has to rearranged or delayed. Such topics involve for example language idiosyncrasies, like

```
public static void main (String[] args)
```

and the usage of overly complex library classes (e.g. input handling) early on.

2.1.2 Avoid exceptions to the general rule

General rules are:

- True objects only
- Control is decentralised, no God-class
- Autonomous objects, encapsulation rules

No concepts must be introduced using flawed examples, but a careful choice of sound classes. Java strings for example are not "real" objects; the `main`-method is not a "real" method and so on.

2.1.3 Objects from the very beginning

Examples involving multiple objects should be used early on. The goal is to make the basic components of solutions and software easy to accept.

2.1.4 OOA&D early

Students should be taught a systematic way to develop software given a problem statement. In FattaOO the choice was made to use CRC cards (Class-Responsibility-Collaboration [3]) Over the years a course-material has been developed, refined and now contains an extended methodology for the CRC approach. The development of Role-Play-Diagrams (RPDs) has greatly improved the possibilities to understand and document the role-playing used to try out an early design [7].

2.1.5 Exemplary examples

All examples used in classes and exercises should comprise well-designed classes that fill a purpose (apart from exemplifying a certain language specific detail). Examples should be non-trivial and involve multiple classes.

2.2 Implications for small examples

Based on the above ideas for the CS1 in Java, a number of difficulties regarding examples have become obvious. Some of these observations are commented on in Holland et.al. [8].

2.2.1 No magic

The use of complex classes to relieve students of complicated I/O, tends to make things happen out of control of the programmer. Make sure that any code supplied is possible to understand for the novice programmer.

2.2.2 Avoid exceptions to a general rule

This has several implications.

- `main` is a non-typical method; no object is created and the method is never explicitly invoked.
- `main` should not be responsible for a lot of work, since this centralises control.
- `String`-objects are non-typical objects; they are immutable and can be created without constructors. Apart from this, they are strongly connected to printing and it is tempting to use `System.out.println` to show the value of the object.

Even if the infamous "Hello world"-example seem to be replaced nowadays, there are still far to many more or less of the same kind:

```
public class Ex1
{
  public static void main(string[] args)
  {
    String s = "Welcome!"; // your first object
    System.out.println(s);
  }
}//Ex1
```

An example of this kind early in the course leaves the lecturer with two options:

- Avoid commenting on the many words in line 3: `public static void main(string[] args)`
- Try to convince the students that they can do without understanding these words, "We will explain these later".

Neither of them especially tempting.

Using `main` to illustrate smaller details like control-statements or the invocation of methods does not implicate the true use if these syntactical constructs.

Normally `main` would be used only for creating the object responsible for the solution of a problem.

```
public class PlayYatzy
{
  public static void main(string[] args)
  {
    Yatzy game = new Yatzy();
    game.start();
  }
}// PlayYatzy
```

This also adds to the complexity and length of the example, since now we are stuck with a much more complicated example, consisting of several classes, some of them with non-trivial behaviour.

Still one might argue *for* using the main-method. One reason for insecurity among CS1 students is the lack of control. A common question is "How is this run?" Dealing with objects,

there is no really good answer to that question. Working with complete applications is one way of gaining a sense of control. “This is a complete program – and I wrote it myself” is a comforting feeling that not should be underestimated.

2.2.3 Objects from the very beginning

It is really difficult to come up with good examples of reasonably realistic objects early on in a course, when not being able to use almost any of the more sophisticated concepts in Java. Strings are common, but are disqualified, for reasons discussed in 2.2.2. Many of the textbook examples consists of single-object situations, such as a class `PigLatinTranslator`, where the reason for more than one object might be difficult to understand. A similar example:

```
        Random generator = new Random();
        int i;

        i = generator.nextInt();
        System.out.println ("A random int: "+i);
        i = generator.nextInt(10);
        System.out.println ("From 0 to 9: " +
i);
```

What is the need for the class `Random`? Is it ever necessary to have more than one object of this kind? The option left is to use the static method `Random` in `Math`.

```
        int i;
        double d;

        d = Math.random();
        i = (int) (d*10);
        System.out.println ("Random : "+i);
        System.out.println ("From 0 to 9: " +
i);
```

One of the benefits of this is that the students need to think about how to adjust the random number to cover the intended range of values. On the other hand we end up with breaking one of the general rules using non-typical methods (a static one). Not to mention the fact that this is completely taken out of context. What kind of object would behave like this? So what would be a good example then? In [12] the `TicketMachine` is introduced. It is a model of a naive ticket machine that issues flat-fare tickets. It is obvious that there will many objects of this kind, physically placed at local bus-stops, and maintained from an administrative system at the bus-company. The criteria stated in [8] are fulfilled by this class and it is easy to show the need for more complex behaviour.

2.2.4 OOA&D early

Being able to think and talk about objects without having to worry about code and syntactical details is rewarding. Still there are difficulties finding suitable problems to work with. They need to be sufficiently simple to grasp, but still reasonably large to justify the approach. One problem is that focusing on OOA&D could be frustrating for both students and lecturers.

There is a trade-off when deciding to use time for OOA&D instead of spending it practising skills. Important is to avoid problems involving to close connection to graphics, such as games. Working with models for a realistic system, such as the core of a library-system or a scheduling system at a cab company has turned out to be suitable for practising object-oriented problem-solving.

2.2.5 Exemplary examples

To illustrate some specific construct in the language, we often make a small example without context, for ex in lab ample to illustrate the for-loop we might use something like this

```
int i = 0;
for (int j=0, j< 10, j++)
{
  i = i+j;
  System.out.println("j="+j+" i="+i);
}
System.out.println("The  sum is " i);
```

To the experienced programmer this is no problem, but to the novice this is confusing. Where "is" this code? What kind of objects would have this kind of behaviour? It is obviously tempting to use printing to show what is happening in the code, but quiet contradictory to how the concept illustrated is supposed to be used "for real".

3 UNDERSTANDING OBJECTS

Moving students to an object-oriented view of problems and solutions seem to be hard or the teaching community. A lot of good work has been done and many ideas tested, three of them described below. But there is still no single approach to help us out.

3.1 Role-Play

The general idea of role-play is to "act" objects physically and by this getting acquainted with the idea of objects being autonomous, though described and constructed by the same "blue-print". This is probably a good idea, but the problem to find the right examples remains. Andrianoff and Levine[2] suggests a role-play consisting of an Acrobat, a Choreographer, an AcrobatWithBuddy, an Author, and a Curmudgeon. This is a sensitive context, since humans normally does things differently, even when asked to perform the same operation. That could lead to the conclusion that objects of the same class can, from an outside perspective, perform the same operation, but solve the task in different ways among themselves. The suggestion for illustrating inheritance with one person (object) "claiming" another person (object) when being created seem inevitably leading to confusion. There will be two persons representing different parts of the same object!

3.2 Using graphics and GUIs

Obviously graphics and visual feedback makes things more fun to explore. This also consistent with the general perception of what Java is all about. Everybody is familiar with Applets and fancy GUI's for the simplest tasks on the web. Why should they expect anything less from a course in "Java"? There are numerous examples of environments and toolkits designed for use in introductory programming courses. Alice [1], BlueJ [5], Jeroo [9] and Power Tools described in [13] are only a few examples.

3.2.1 Advantage - fun

There is no doubt that "fun" enhances learning. The environments mentioned above are fun to work with, even for an experienced programmer. Who has not spent hours trying to make some tiny detail of a silly program even better, just for the fun of it? Programming IS fun!

3.2.2 Disadvantage – "not within my ability"

Supplying the graphical environment, as in [13] is no doubt a better way than having the students struggle with the graphics themselves. Modern software is what we are aiming for and this by all means excludes ASCII-based I/O through some line-based media. All modern applications use a nice GUI for sure. The big disadvantage is that many of the students tend to be discouraged, because they soon enough realise that they are nowhere near the ability to create anything like this on their own. If the ambition is to liberate the students from the gory details of graphics and window handling, we end up with lots of code supplied, not possible for the students to understand.

3.2.3 Disadvantage – too much magic!

If supplying the graphics through toolkits is out of the question, but graphics still motivated by the argument of being fun, we have to let the students do it themselves. Talking about magic… One obvious disadvantage is that it can be very tricky to get the graphics to work in Java. In our experience there is a huge probability that the students get stuck focusing on syntactical details concerning graphical details, rather then solving the real problem given. Working with graphical components tends to create more problems then they solve. Thereby consuming a lot of time, taken from more important aspects of problem-solving and programming.

3.2.4 Disadvantage - confusion

Another aspect of this is that there seem to be a cognitive difficulty to differ between the visual representation of an object and the object itself. This might actually lead to misconceptions that are very hard to deal with. Another disadvantage of supplying code is that students tend to have a difficulty separating which components that belong to the language and which are supplied by libraries. Inheritance is intense in the AWT and makes it even harder to understand the behaviours of the objects involved.

3.3 CRC with Role-Play-Diagrams (RPDs)

Introducing OO-analysis and design immediately when following the objects-first approach makes the need for a methodology obvious. But is there one? No, not so far. So the best we can do is to try to enhance the student's notion of good classes and good design through practical experience. After working with the CRC-approach for some years, we found it necessary to develop a way to document the scenarios used to try the design for a system. We found out that calling the brainstormed potential classes "candidate-objects" as done in [3], was misleading. Furthermore we discovered that the use of cards representing both the class and a single object during the recording of a scene was even more confusing. Another shortcoming of the method was the lack of notation for documenting the scenes as they were used to test the system-design. With simple post-it notes and paper and pen (or whiteboard) we developed the notation presented in [7]. Post-it notes are used to represent objects, while cards are used to keep track of the classes. The order of messages and actions are labelled sequentially and noted on the paper, thereby keeping record of the scene. This makes it simple to have many objects; some may be created as a result of actions resulting from messages.

4 THE "who-am-I"-PROBLEM

Apart from the problems mentioned above there is another detail confusing students. When introducing programming lecturers switch between different views of "I".

- "I" am the one executing the code/program
- "I" am the constructor of the code/class/program
- "I" am the user of the object/program

This is highly confusing to a novice. When designing a class it seem like a good idea to think about how the objects are going to be used, which behaviours we want to make available to the "user". This makes us switch back and forth between these views. This might cause the students to believe that objects change state and behaviour by editing the source-code of the class. Then when tracing the code to see if it works we switch perspective once more.

5 CONCLUDING REMARKS

Looking at experiences made, and described in this paper, it is easier to state what makes a bad example than a good one. Being able to categorize favourable characteristics would certainly aid in designing "good" examples. So this will be interesting to investigate.

This page intentionally left blank

6 REFERENCES

[1] Alice home page http://www.alice.org/

[2] Andrianoff S.K. and Levine D.B. "Role playing in an object-oriented world" In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education (SIGSCE'02)* Cincinnati, Kentucky, 2002 pp 121-125

[3] Bellin, D. and Simeone, S. *The CRC Card Book*, Addison-Wesley, 1997.

[4] Joseph Bergin *The Object Game – An Exercise for Studying Objects* Webpage http://wol.pace.edu/~bergin/patterns/objectgame.html

[5] BlueJ home page http://www.bluj.org/

[6] Börstler J, Johansson T. and Nordström M. "Teaching OO concepts—a case study using CRC-cards and BlueJ". In *Proceedings of the Frontiers in Education conference (FIE2002)* Boston, USA November 6-9, 2002

[7] Börstler J. *Object-Oriented analysis and Design Through Scenario Role-Play*. Technical Report UMINF 04.04, ISSN-0348-0542 Umeå University, Umeå Sweden 2004.

[8] Holland S., Griffiths R. and Woodman M. In *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education,* San Jose, California, USA 1997 pp. 131-134

[9] Jeroo home page http://www.jeroo.org/

[10] Kölling, M. and Rosenberg, J., "Guidelines for Teaching Object Orientation with Java", In *Proceedings ITiCSE'01*, Canterbury, UK, June 2001, pp. 33-36

[11] Barnes D. J. and Kölling M. *Objects First with Java - A Practical Introduction using BlueJ*, 2nd ed. Pearson Education, 2004

[12] Nevison C. and Wells, B. "Using a Maze Case Study to Teach Object-Oriented Programming and Design Patterns", In *Proceedings ACE2004*, Dunedin, NZ, pp. 207-215

[13] Prolux V., Raab J. and Rasala R. "Objects from the Beginning-With GUIs", In *Proceedings ITiSCE'02*, Aarhus, Denmark, June 2002, pp. 65-69